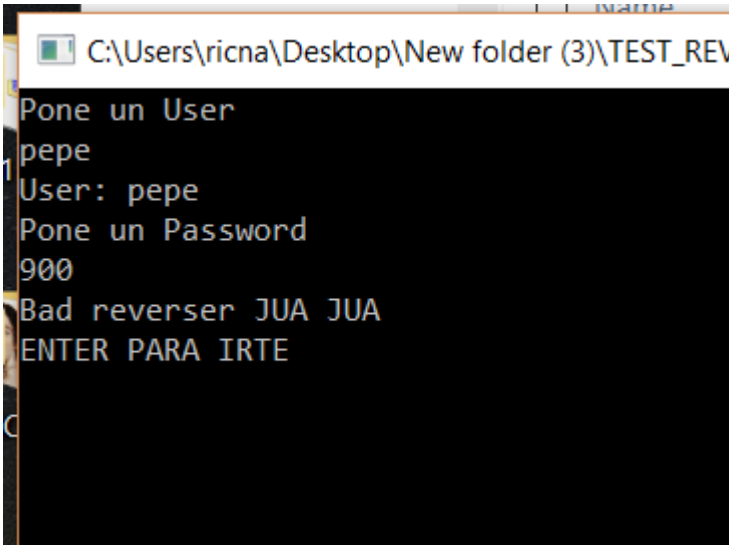
Введение в реверсинг с нуля используя IDA PRO. Часть 12.

Хорошо, чтобы не было скучно, мы будем объединять теорию и некоторые упражнения, в этом случае, это другая программа скомпилированная мной, которая называется ***TEST_REVERSER.EXE*** и которая очень простая, но она поможет нам увидеть некоторые новые вещи в статическом реверсинге и которые мы проверим в отладчике.

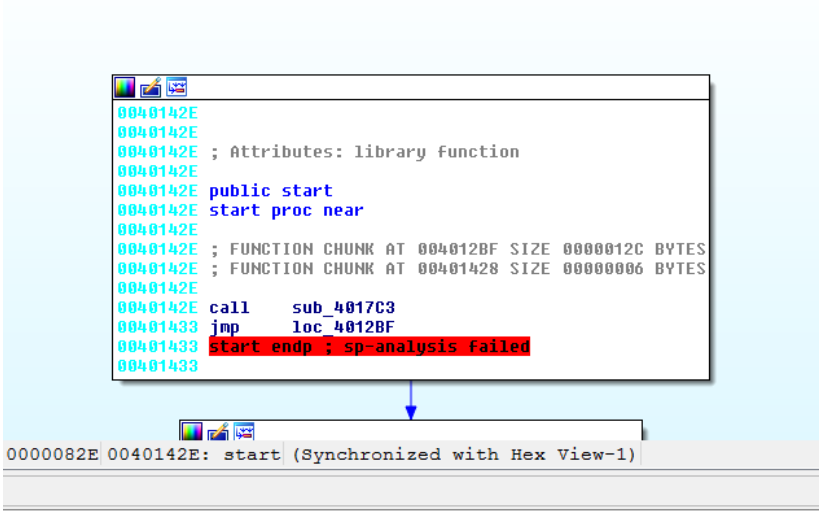
Если мы запустим программу вне **IDA**, то мы увидим.



Нас просят ввести имя пользователя и затем пароль, а потом программа говорит нам, что мы проиграли, и она смеётся над нами.

Давайте откроем её в **IDA**, чтобы увидеть программу в статическом виде.

Поскольку я не использую символы, всё оказывается слегка уродливым.



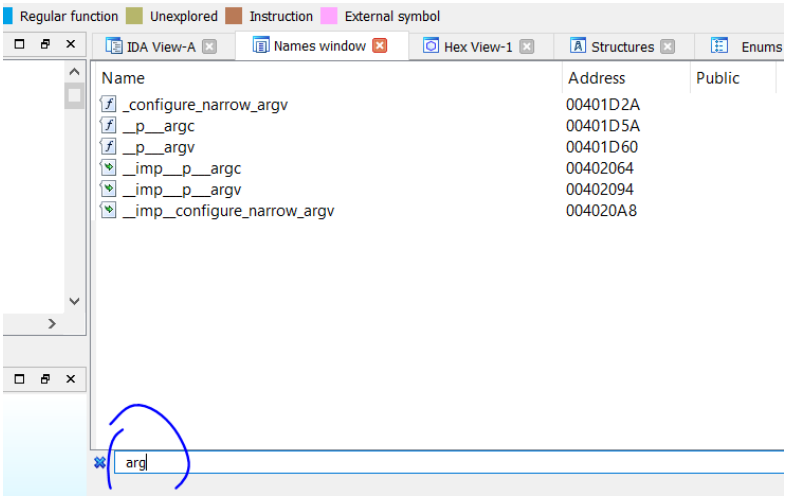
Очевидно, она не открывается в функции ***MAIN***, зато открывается в ***EP***, но хорошо, действительность почти всегда такова и мы разберемся с этой проблемой.

Один из способов, который мы уже видели, чтобы попасть в `*горячую часть программы*` - найти строки, мы уже знаем как это делать, также в этих консольных программах на ***C++***, это один из способов найти ***MAIN*,** который почти всегда работает.

Мы знаем, что в функцию передаются аргументы ***ARGC, ARGV*** и т.д. Это аргументы консоли.

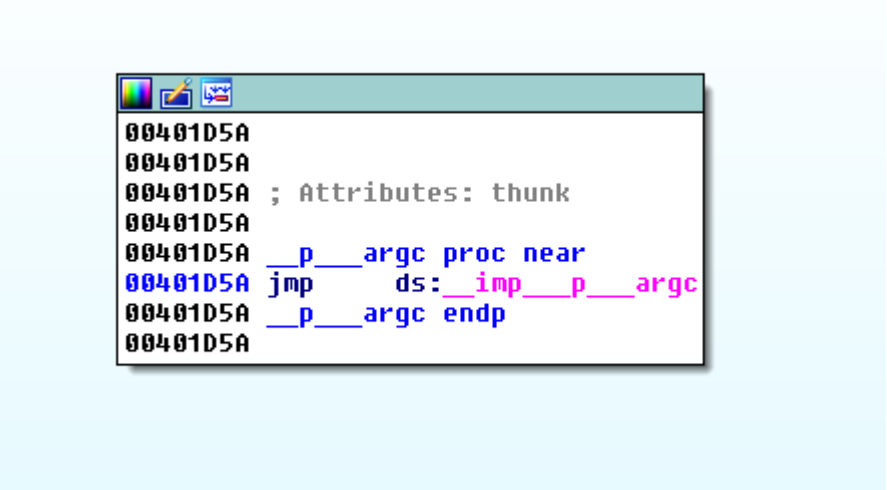
INT MAIN(INT ARGC, CHAR *ARGV[])

Мы уже видели в предыдущем примере, что даже если аргументы не используются, они всё равно ***ПОМЕЩАЮТСЯ*** в стек, иногда это действует по умолчанию для консольных исполняемых файлов, поэтому мы можем искать их во вкладке **NAMES**, чтобы увидеть, есть ли они там или нет.

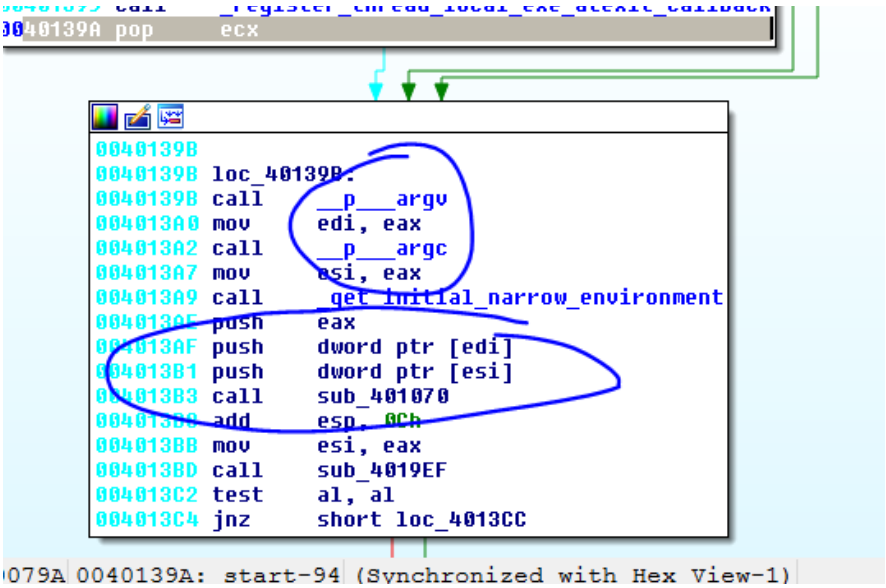


Отсюда и далее, когда я говорю - "*На вкладке **XXX***", Вы уже должны знать, что это открывается в меню **VIEW → OPEN SUBVIEW → XXX**, чтобы не повторять это более.

Здесь с помощью комбинации **CTRL + F** мы фильтруем ввод, введём например ***ARG*** и мы увидим записи, давайте сделаем двойной щелчок по ***_P_ARGC***.

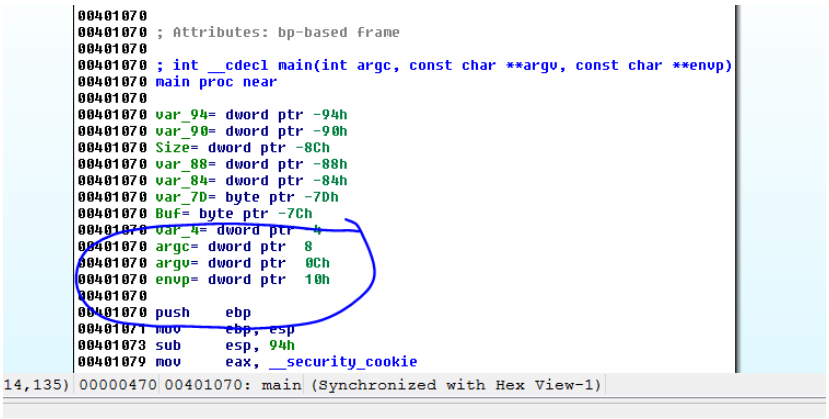


Ища ссылки с помощью **X** мы находим

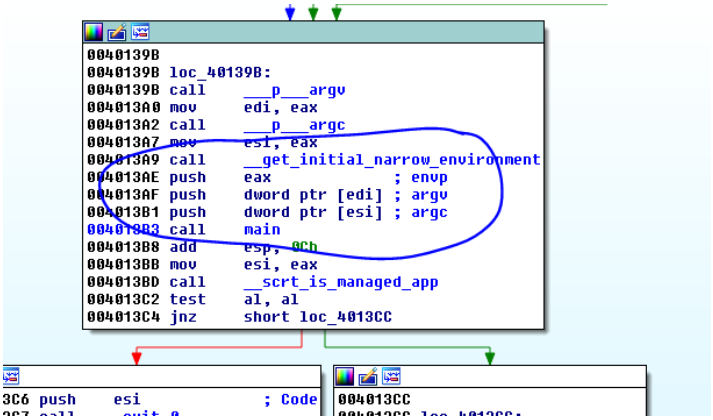


Здесь мы видим как программа вызывает функции ***_P_ARGV*** и ***_P_ARGC*** и то, что она передает содержимое результата в функцию ***MAIN***, адрес которой в этом случае - **0x401070**.

Если я посмотрю ту же самую функцию в моей **IDA,** в версии с символами.

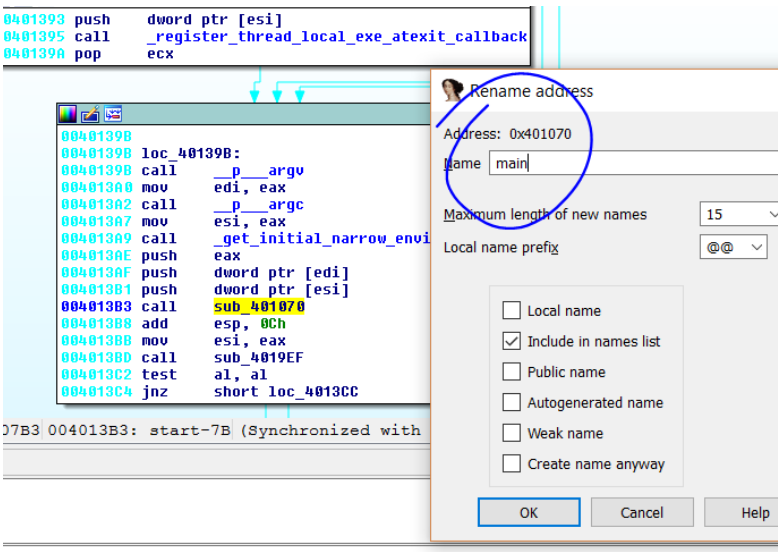


И мы видим ссылку.

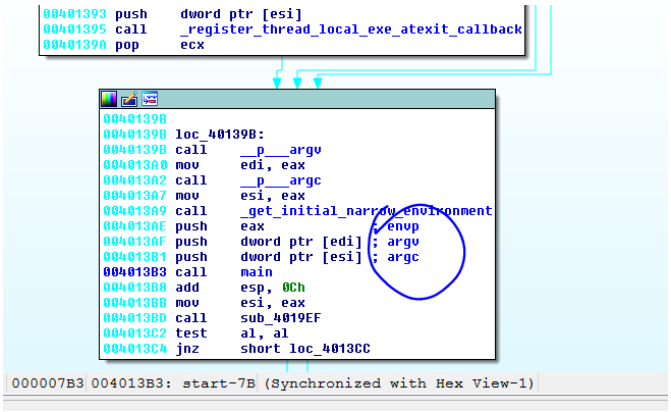


Очевидно это не хорошая идея использовать такие читы, так можно только проверить, что метод для нахождения ***MAIN*** работает и здесь - это полностью верно, ища ссылки аргументов, которые передаются консолью, мы прибываем к ***MAIN***.

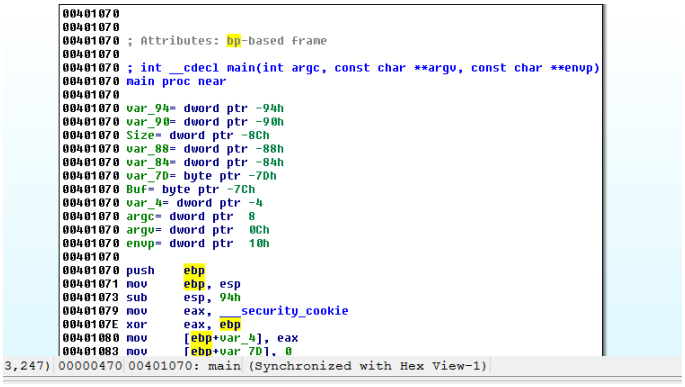
Давайте переименуем этот вызов в ***MAIN***.



IDA автоматически переименовывает ***АРГУМЕНТЫ*** узнав, что вышеупомянутая функция – это ***MAIN***.



Сейчас, это больше похоже на версию с символами.



Мы видим в этом случае, что переменные и аргументы более нумерованы чем в предыдущем примере.

Если мы сделаем двойной щелчок на любой переменной или аргументу, мы увидим статическое представление стека.

```
-00000013                   db ? ; undefined
-00000012                   db ? ; undefined
-00000011                   db ? ; undefined
-00000010                   db ? ; undefined
-0000000F                   db ? ; undefined
-0000000E                   db ? ; undefined
-0000000D                   db ? ; undefined
-0000000C                   db ? ; undefined
-0000000B                   db ? ; undefined
-0000000A                   db ? ; undefined
-00000009                   db ? ; undefined
-00000008                   db ? ; undefined
-00000007                   db ? ; undefined
-00000006                   db ? ; undefined
-00000005                   db ? ; undefined
-00000004 var_4             dd ?
+00000000  s                db 4 dup(?)
+00000004  r                db 4 dup(?)
+00000008 argc              dd ?
+0000000C argv              dd ?                    ; offset
+00000010 envp              dd ?                    ; offset
+00000014
+00000014 ; end of stack variables
```

Мы смотрим снизу вверх и видим, что логически первыми идут аргументы функции, которые будут всегда под адресом возврата ***R,*** так как они передаются через **PUSH** и они сохраняются в стек перед вызовом функции с помощью **CALL**, которая затем сохраняет адрес возврата в стек.

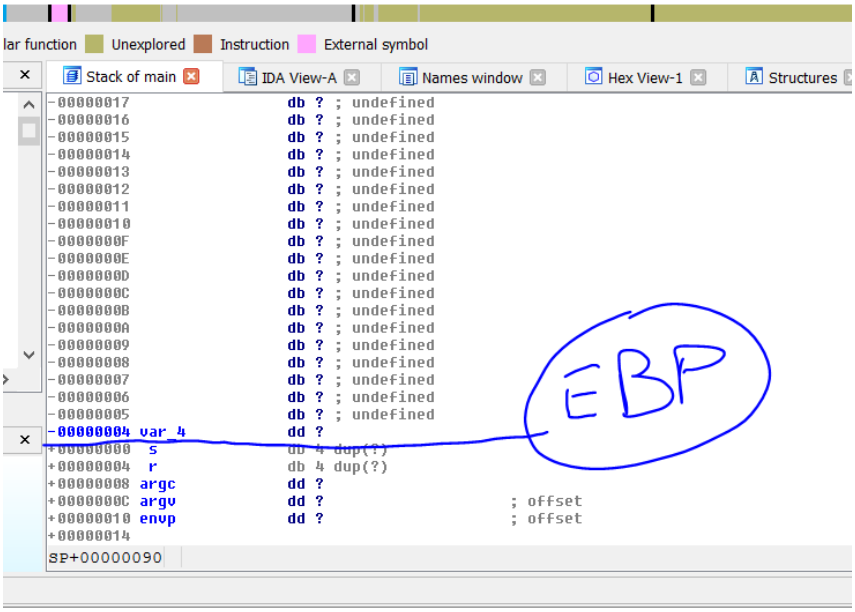
Затем у нас идёт ***S*** или что, то же самое, что и ***STORED*** **EBP**, который является **EBP** функции, которая вызывается функцией ***MAIN***, он сохраняется в стек, когда начинает выполняться функция с помощью инструкции **PUSH EBP**.

```
00401070 var_94= dword ptr -94h
00401070 var_90= dword ptr -90h
00401070 Size= dword ptr -8Ch
00401070 var_88= dword ptr -88h
00401070 var_84= dword ptr -84h
00401070 var_7D= byte ptr -7Dh
00401070 Buf= byte ptr -7Ch
00401070 var_4= dword ptr -4
00401070 argc= dword ptr  8
00401070 argv= dword ptr  0Ch
00401070 envp= dword ptr  10h
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 sub     esp, 94h
00401079 mov     eax, ___security_cookie
0040107E xor     eax, ebp
00401080 mov     [ebp+var_4], eax
00401083 mov     [ebp+var_7D], 0
```

Затем программа помещает **ESP** в **EBP**, помещая значение в **EBP**, которое оно будет иметь в этой функции, чтобы быть ***БАЗОЙ*** откуда берутся аргументы, которые ниже ***БАЗЫ*** и локальные переменные, которые выше неё, и наконец инструкция **SUB ESP, 0x94** сдвигает **ESP** выделяя место для переменных и локальных буферов, поэтому они выше, в этом случае размер для буфера будет равен **0x94**, потому что компилятор вычисляет, размер который ему нужен, чтобы

зарезервировать место для переменных и буферов, в соответствии с тем, как мы запрограммировали нашу функцию.

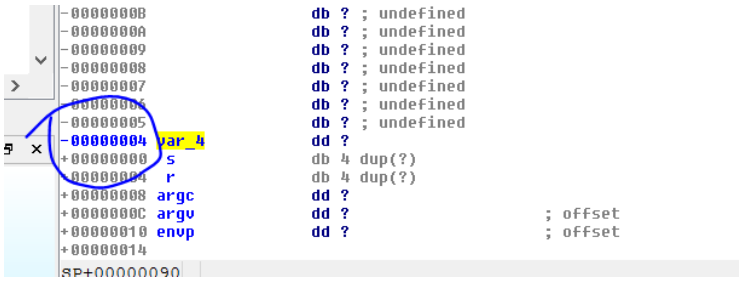
ESP остается со значением выше этого зарезервированного пространства для локальных переменных и **EBP** указывает на ***БАЗУ*** или ***ГОРИЗОНТ***, который делит стек на локальные переменные, которые выше и ***СОХРАНЕННЫЙ EBP***, ***АДРЕС ВОЗВРАТА*** и ***АРГУМЕНТЫ***, которые ниже него.



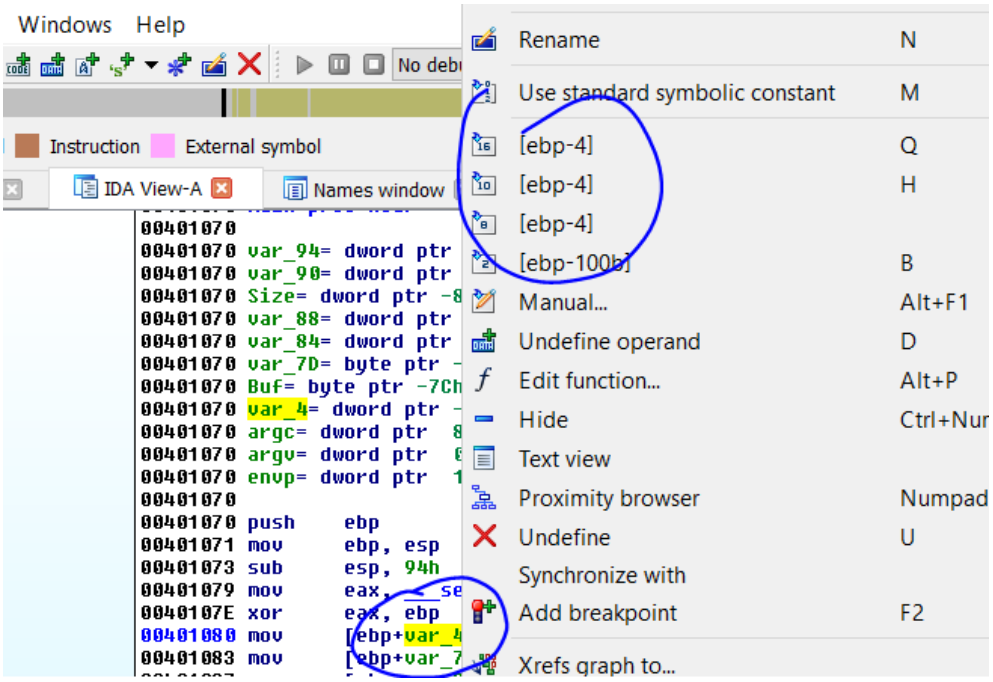
Вот почему в функциях основанных на **EBP**, как только программа вызывает функцию, она сохраняет с помощью инструкции **PUSH EBP** значение **EBP** функции, которую вызывает программа, затем программа помещает **ESP** в **EBP** и это считается теперь как горизонт, вот почему в статическом представлении стека, **IDA** показывает **000000000** как горизонт, а выше видно знаки минус (**-**), а ниже знаки плюс (**+**).

Вот почему ***VAR_4*** имеет значение **- 00000004**, потому что переменная берет **EBP** как ***БАЗУ*** или как **0** и математическим адресом переменной будет **EBP - 4**.

И ниже ***ARGC*** будет равен **EBP + 8**, это видно по колонки слева.

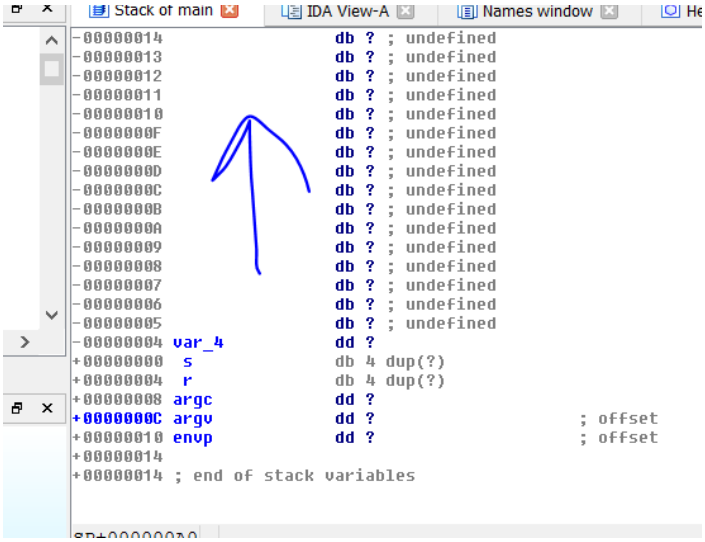


Это можно проверить это в листинге, внутри функции ***MAIN*** где программа использует переменную ***VAR_4,*** если мы сделаем правый клик на ней.

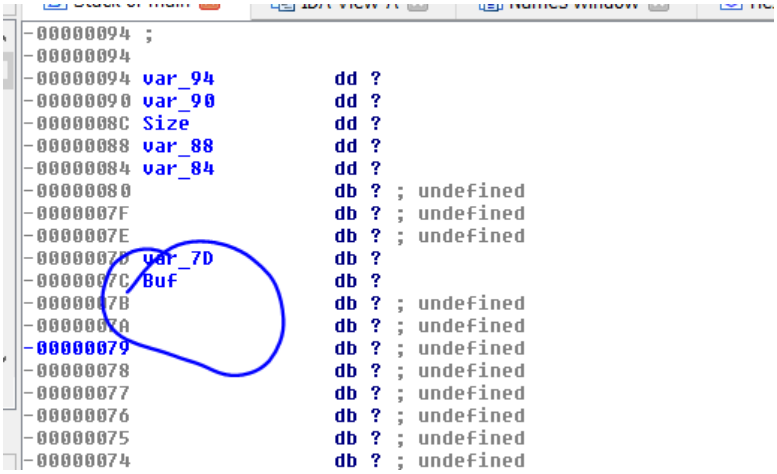


Вернемся к статическому представлению стека.

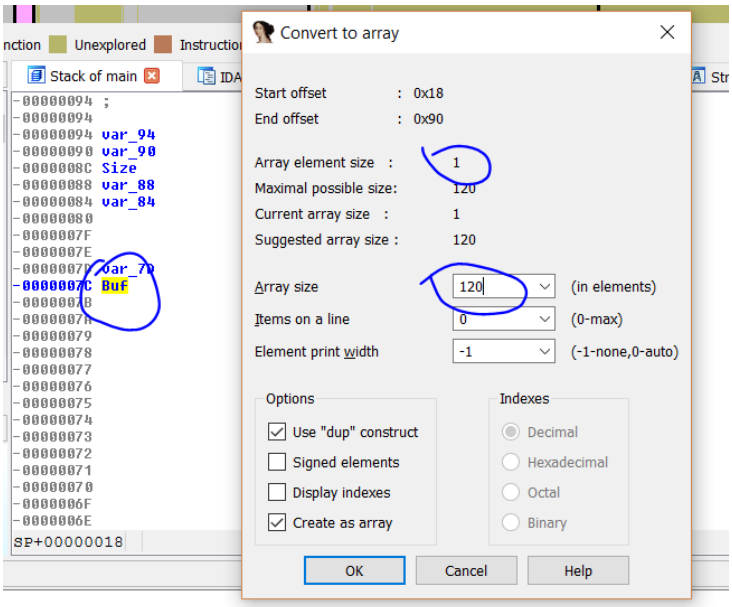
Когда мы видим здесь пустое пространство где нет смежных переменных, возможно это потому что выше есть ***БУФЕР*** (позже мы увидим случаи, в котором пустое пространство является структурой). Сейчас давайте немного поднимемся.



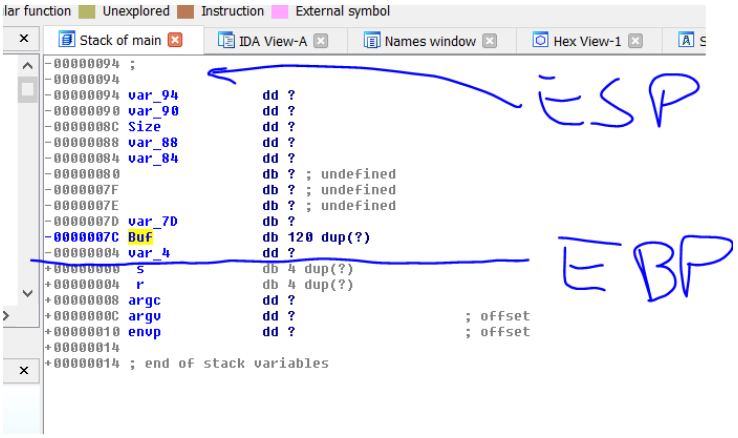
Здесь мы видим переменную ***BUF***, которая является первой переменной над пустой зоной, мы делаем правый клик и выбираем **ARRAY**.



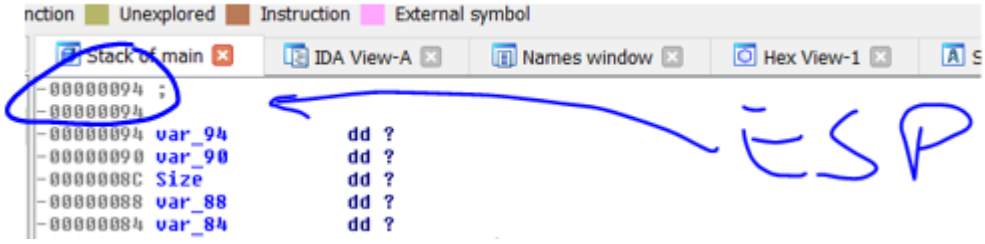
Мы видим, что размер ***МАССИВА*** равен **120** байт, потому что он состоит из **120** элементов по **1** байту.



Сейчас представление стека стало лучше.



Мы видим базу **EBP** и мы помним, что как только **EBP** и **ESP** равны это равносильно инструкции **MOV EBP, ESP**, затем программа вычитает из **ESP** значение **0x94** и **ESP** теперь начинает работать выше зоны для локальных переменных.



Здесь мы видим область в которой **ESP** будет после инструкции **SUB ESP, 0x94**.

Здесь в левой стороне видно значение **-00000094** или что также равно **ESP = EBP – 094**, очевидно затем значение будет продолжать расти по мере работы программы, между другими подпрограммы и т.д., но всегда пока значение находится внутри этой функции ***MAIN*** и пока значение не выйдет из этой области, оно будет продолжать работать в области не выше **0x94** зарезервированную часть для переменных, чтобы не наступить на них.

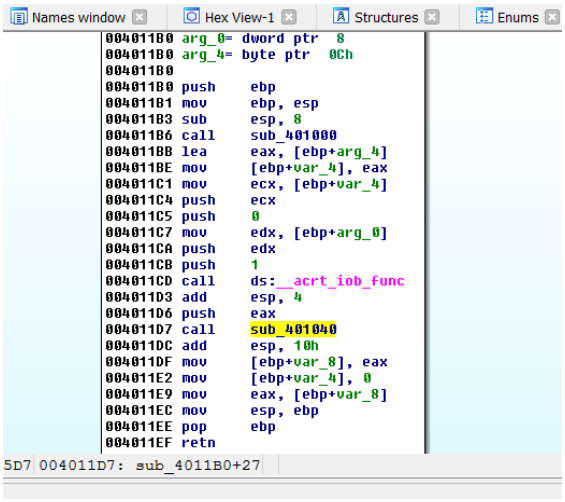
Хорошо, однажды мы уже видели статическое представления стека, давайте отреверсим переменные, поскольку аргументы нам известны (***ARGC***, ***ARGV***, и т.д.)

```
00401070 var_7D= byte ptr -7Dh
00401070 Buf= byte ptr -7Ch
00401070 var_4= dword ptr -4
00401070 argc= dword ptr  8
00401070 argv= dword ptr  0Ch
00401070 envp= dword ptr  10h
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 sub     esp, 94h
00401079 mov     eax, ___security_cookie
0040107E xor     eax, ebp
00401080 mov     [ebp+var_4], eax
00401083 mov     [ebp+var_7D], 0
```

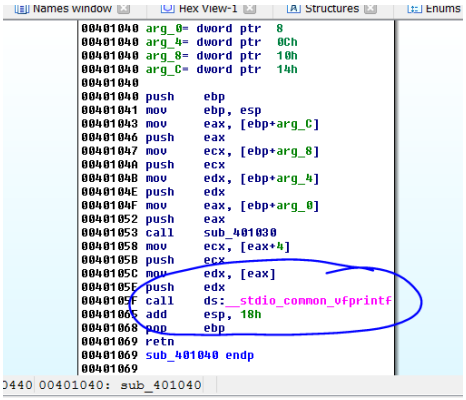
Мы уже видели, что ***VAR_4*** это переменная ***COOKIE_SEGURIDAD*** или ***CANARY***, мы видим, что инструкция считывает это значение, затем ***XORum*** его с помощью **EBP** и сохраняет его в стек, чтобы защитить стек от ***ПЕРЕПОЛНЕНИЯ***, поэтому давайте переименуем эту переменную.

```
00401070 Size= dword ptr -8Ch
00401070 var_88= dword ptr -88h
00401070 var_84= dword ptr -84h
00401070 var_7D= byte ptr -7Dh
00401070 Buf= byte ptr -7Ch
00401070 COOKIE_SEGURIDAD= dword ptr -4
00401070 argc= dword ptr  8
00401070 argv= dword ptr  0Ch
00401070 envp= dword ptr  10h
00401070
00401070 push    ebp
00401071 mov     ebp, esp
00401073 sub     esp, 94h
00401079 mov     eax, ___security_cookie
0040107E xor     eax, ebp
00401080 mov     [ebp+COOKIE_SEGURIDAD], eax
00401083 mov     [ebp+var_7D], 0
00401087 mov     [ebp+var_88], 0
00401091 mov     [ebp+Size], 8
0040109B push    offset aPoneUnUser ; "Pone un User\n"
004010A0 call    sub_4011B0
004010A5 add     esp, 4
004010A8 mov     eax, [ebp+Size]
004010AE push    eax             ; Size
004010AF lea     ecx, [ebp+Buf]
004010B2 push    ecx             ; Buf
153) (499,242) 00000480 00401080: main+10
```

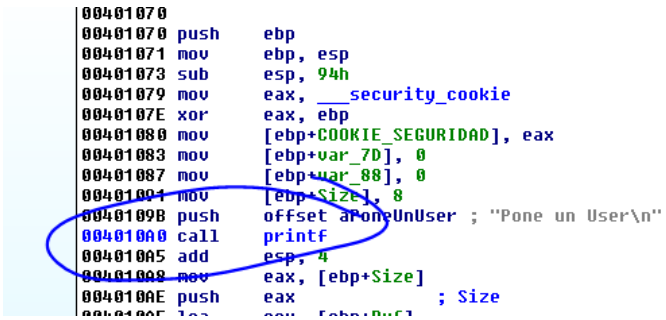
Равно как и в предыдущем примере ***API*** функция ***PRINTF*** не имеет символов и она не отображается, но я наблюдаю строки для неё, которые она печатает в консоли и мы видим эту функцию по адресу **0x4011B0**.



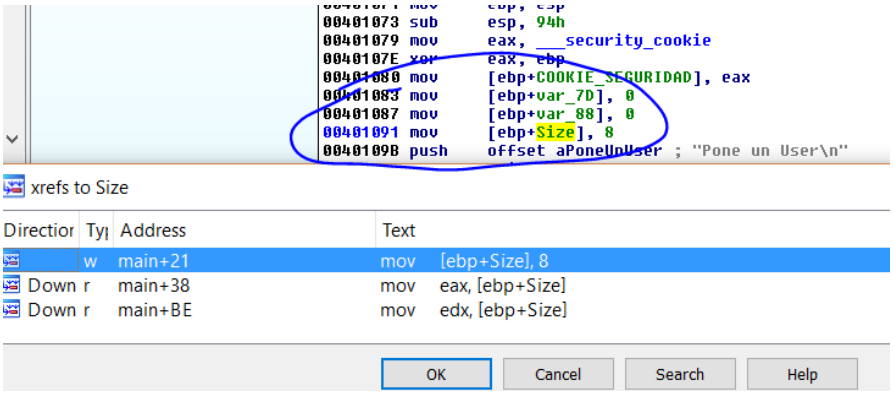
И здесь внутри по адресу **0x401040** мы видим.



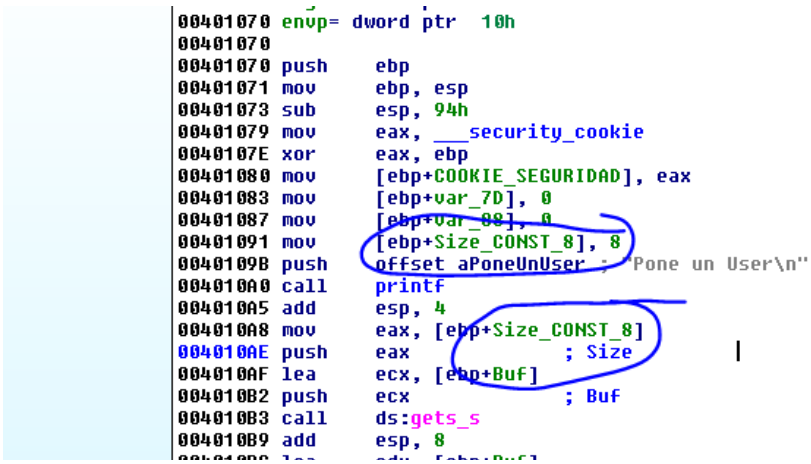
Так что, давайте переименуем функцию по адресу **0x4011B0** в функцию с именем ***PRINTF***.



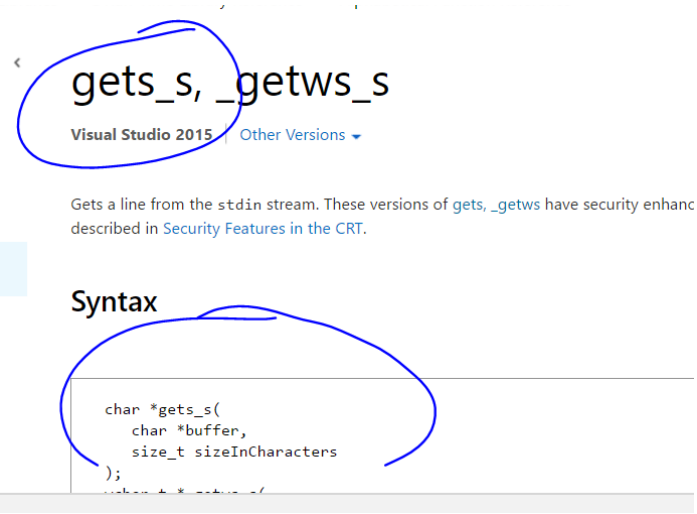
Давайте идти дальше.



Мы видим, что размер переменной инициализируется с помощью числа **8** и больше это значение никогда не изменяется, есть только два чтения среди следующих ссылок, поэтому мы переименуем размер этой переменной в имя ***_CONST_8***.

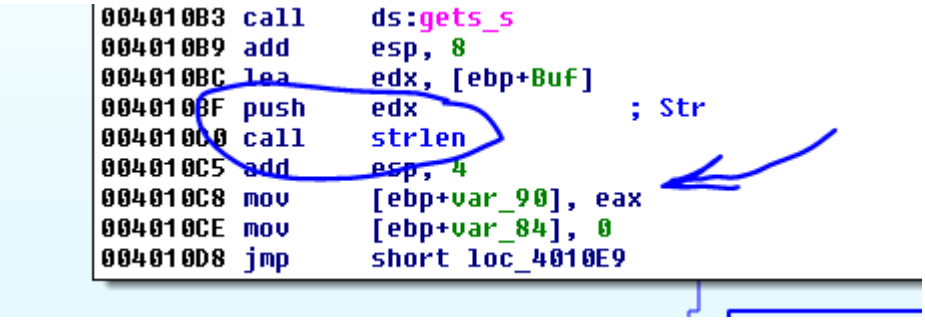


Затем мы видим вызов функции ***GETS_S***, которая является эволюцией функции ***GETS*****,** но с ограничение по количеству вводимых символов, которые мы можем ввести(это такая защита), в этом случае максимальным значением будет **8,** которое помещается в **EAX** и передается как аргумент с помощью **PUSH EAX**, а затем **LEA** получает адрес переменной ***BUF*** или ***BUFFER***.

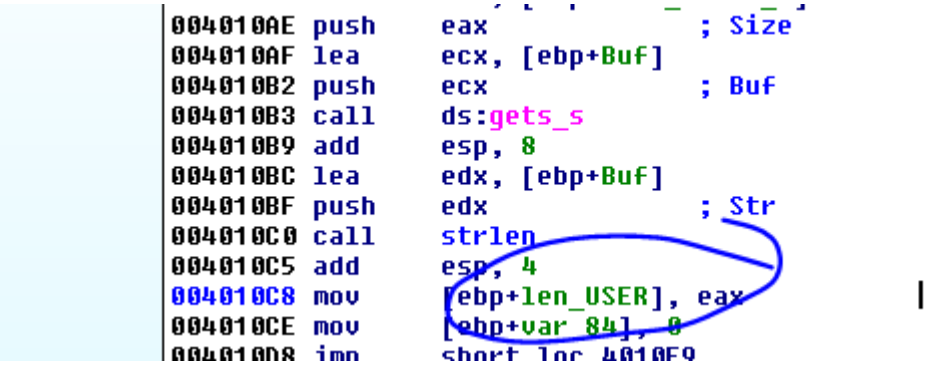


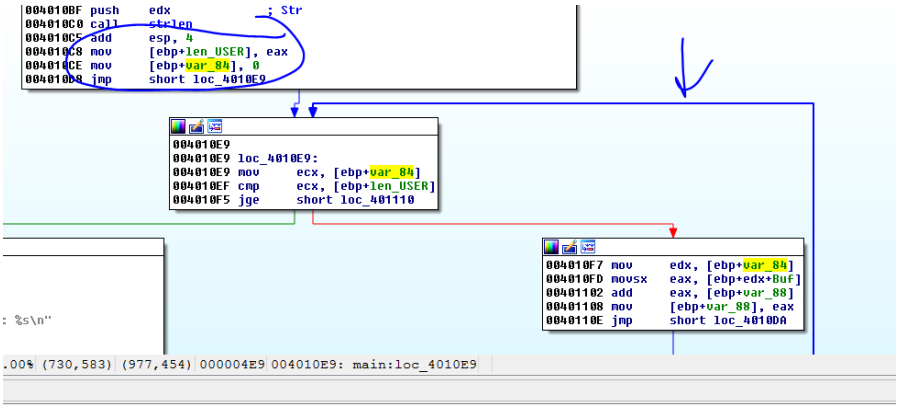
Конечно, если мы введём больше символом чем **8** и нажмём **ENTER**, функция также будет обрезать ввод и случится возврат.

Так что мы знаем, что в ***BUF*** будет помещаться имя ***ПОЛЬЗОВАТЕЛЯ***, которое мы набрали и что оно будет иметь максимум **8** символов.

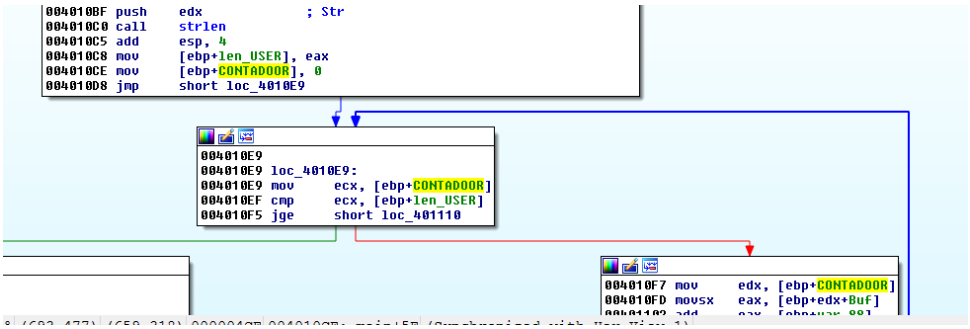


Здесь мы видим, что потом программа передаёт с помощью **PUSH EDX** адрес буфера снова, как аргумент к ***API*** функции ***STRLEN***, чтобы получить длину строки, которая сейчас находится в ***BUF*** и соответствует введенному ***ПОЛЬЗОВАТЕЛЮ***, и сохраняет длину в переменной ***VAR_90*** через регистр-результат **EAX**, так что мы переименовываем ***VAR_90*** в ***LEN_USER***.

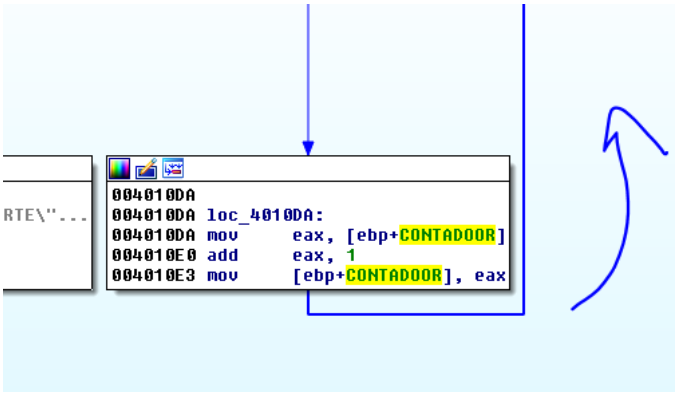




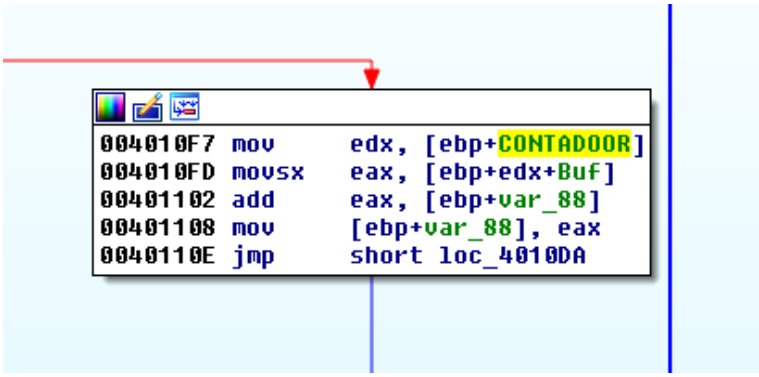
Синяя стрелка всегда указывает на переход назад, который может быть ***ЦИКЛОМ***, по адресу **0x4010CE** программа инициализирует счетчик ***LOOP VAR_84,*** мы также видим, что по адресу **0x4010F5** находится условный переход, который оценивает условие выхода из цикла, счетчик начинается с ***НУЛЯ***, и он будет увеличиваться в каждом цикле, и программа выйдет из цикла, когда счетчик будет больше или равен длине, которую мы ввели в ***LEN_USER***.



Счетчик увеличивается к концу ***ЦИКЛА*** здесь.



Здесь он помещает значение ***СЧЁТЧИКА*** в **EAX** увеличивая его и затем снова сохраняет.



Здесь программа помещает первый байт ***БУФЕРА*** из **EBP + EDX + *BUF*** в **EAX**, поскольку **EBP + *BUF*** складывается со ***СЧЕТЧИКОМ***, который сейчас равен нуль, то он будет увеличиваться пока работает ***ЦИКЛ***, мы видим, что здесь будет складываться все значения символов, которые я набираю, поэтому переменная ***VAR_88*** которая начинается с нуля, будет складываться в каждом цикле ***HEX*** значения каждого символа строки ***БУФЕРА***.

Мы видим инструкцию, которую мы до этого даже ещё не видели - **MOVSX**.

MOVSX И **MOVZX.**

Обе инструкции берут байт и помещают его в регистр, в случае с **MOVZX** заполняются с помощью нулей старшие байты, в то время как в случае с **MOVSX** учитывается знак байта, если он положительный или меньше или равен **0x7F** он заполняется с помощью нулей и если он отрицательный или равен **0x80** или больше заполняется с помощью **0xFF**.

MOVZX EAX, [XXXX]

Если содержимое **XXXX** будет равно **0x40, EAX** будет равен **0x00000040**.

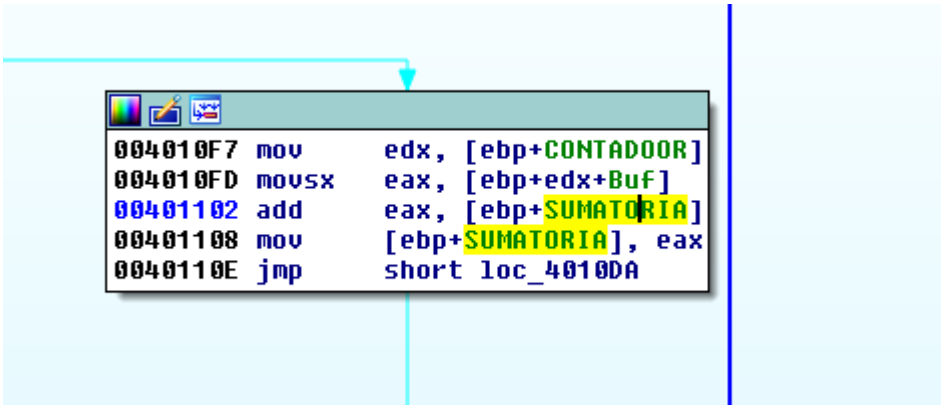
Также например существует инструкция **MOVZX EAX, CL.**

Этот случай похожий, инструкция будет брать значение байта и заполнит с помощью нулей старшие байты.

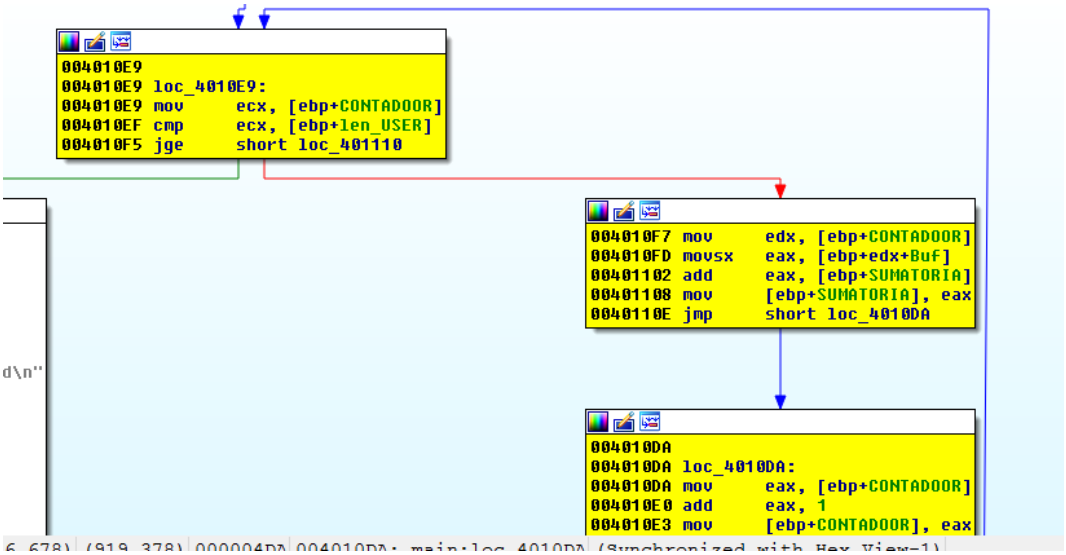
MOVSX EAX, CL

Инструкция принимает во внимание знак байта, если **CL** - например равен **0x40**, **EAX** будет равен **0x00000040** и если бы он был бы равен **0x85**, в этом случае, поскольку у него отрицательный знак и это значение отрицательное **EAX** будет равен **0xFFFFFF85**.

Также, так как мы вводим символы букв и чисел через консоль, эти символы являются положительные ***HEX*** значениями, так что у нас не будет никаких проблем, программа будет складывать значения один за одним и сохранять.

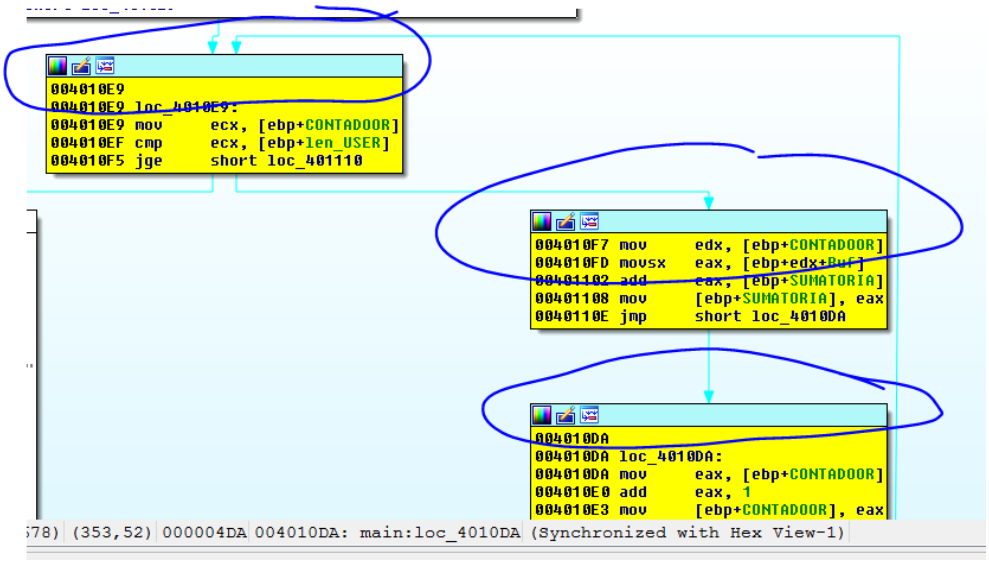


Мы видим, что ***ЦИКЛ*** - это сложение символов, мы покрасим их тем же цветом.

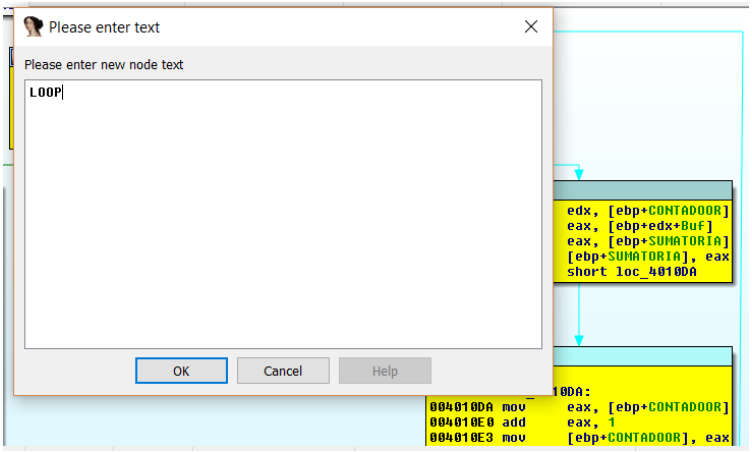


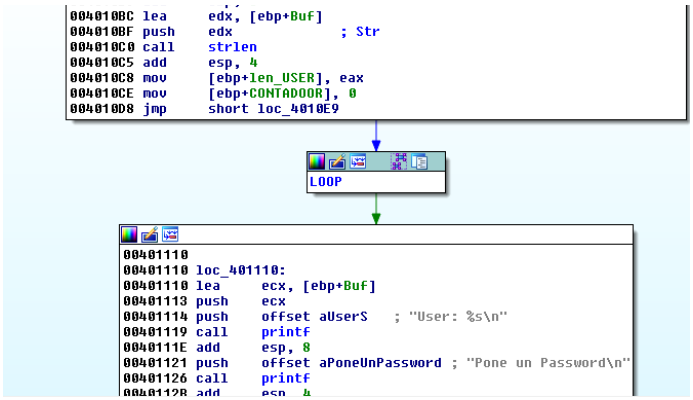
Также я немного увеличил их, перетащив и отпустив нижний блок немного выше.

Есть люди, которые, чтобы убрать блоки с экрана, если они хотят, чтобы они их не беспокоили, группируют их, с помощью нажатия **CTRL** и делая клик в верхнюю панель каждого блока.



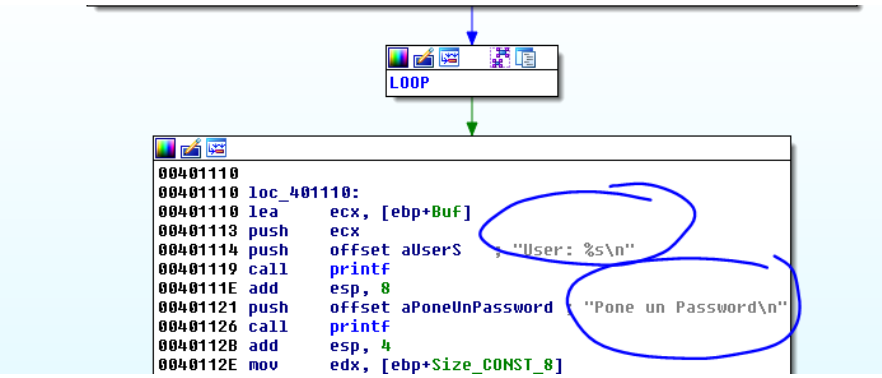
Делаем правый клик и выбираем **GROUP NODES,** а затем вводим имя, например ***LOOP***.





Последнее, если Вы хотите увидеть блоки, которые спрятаны, то это можно сделать через **UNGROUP NODES**.

Затем программа выводит слово ***ПОЛЬЗОВАТЕЛЬ*** и говорит, что нужно ввести ***ПАРОЛЬ***.



Затем программа вызовет снова функцию ***GET_S*** используя тот же самый буфер и тоже максимальное значение вводимых символов.

```
00401126 call    printf
0040112B add     esp, 4
0040112E mov     edx, [ebp+Size_CONST_8]
00401134 push    edx             ; Size
00401135 lea     eax, [ebp+Buf]
00401138 push    eax             ; Buf
00401139 call    ds:gets_s
```

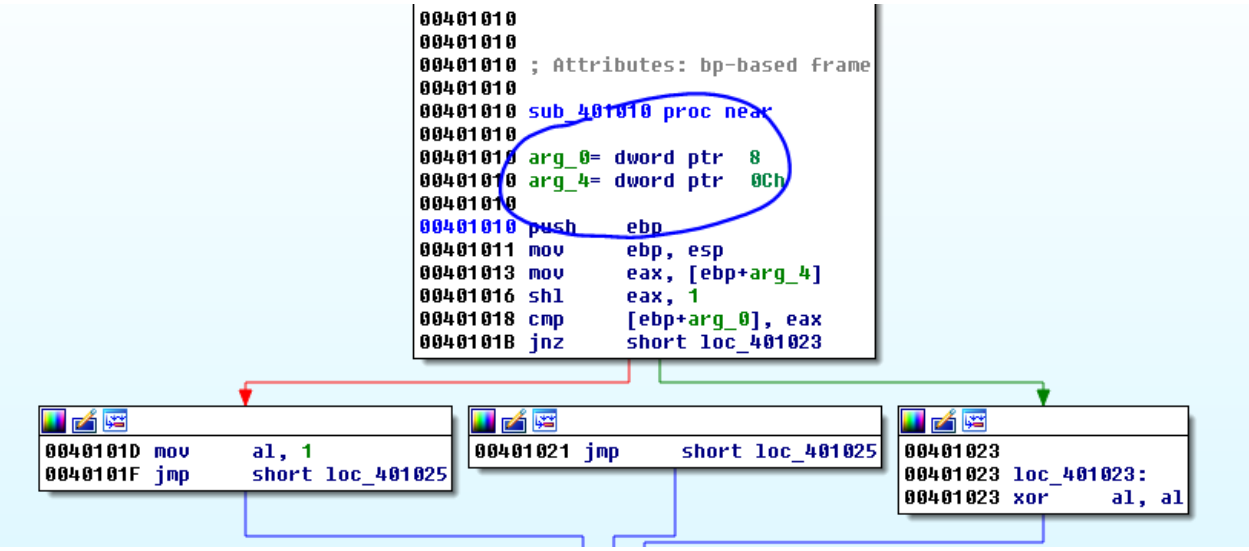
Программа может повторно использовать тот же ***БУФЕР*** для ***ПАРОЛЯ***, в любом случае программа полностью рассчитала ***СУММУ HEX*** значений символов ***ПОЛЬЗОВАТЕЛЯ*** и она больше не будет использовать строку ***ПОЛЬЗОВАТЕЛЬ***.

```
0040113F add     esp, 8
00401142 lea     ecx, [ebp+Buf]
00401145 push    ecx             ; Str
00401146 call    ds:atoi
0040114C add     esp, 4
0040114F mov     [ebp+var_94], eax
```

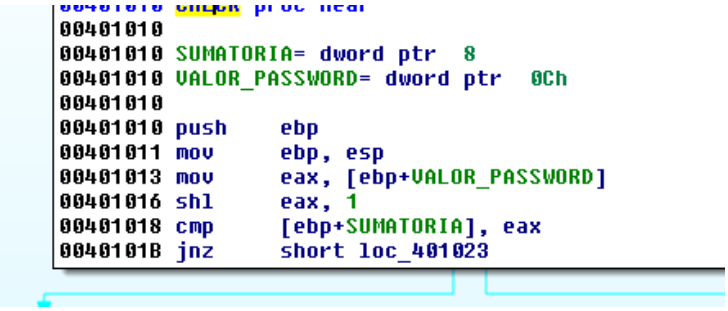
Сейчас она возьмёт ***ПАРОЛЬ*** и преобразует его в ***HEX*** как в предыдущем примере используя ***ATOI***.

```
00401145 push    ecx             ; Str
00401146 call    ds:atoi
0040114C add     esp, 4
0040114F mov     [ebp+VALOR_PASSWORD], eax
00401155 mov     edx, [ebp+VALOR_PASSWORD]
0040115B push    edx
0040115C mov     eax, [ebp+SUMATORIA]
00401162 push    eax
00401163 call    sub_401010
00401168 add     esp, 8
```

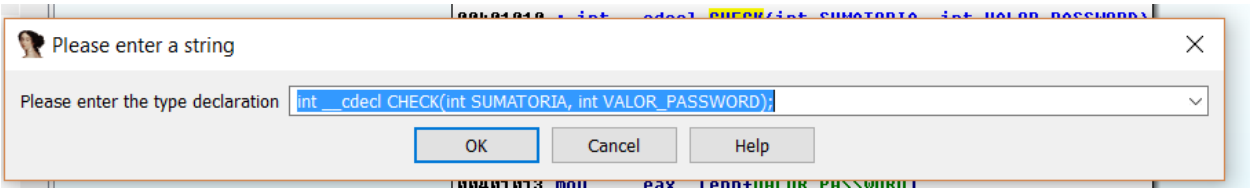
Здесь передаётся значение ***VALOR_PASSWORD*** с помощью инструкции **PUSH EDX** и суммируется с помощью **PUSH EAX**, это будут два аргумента, которые передаются в функцию по адресу **0x401010**, давайте введём их.



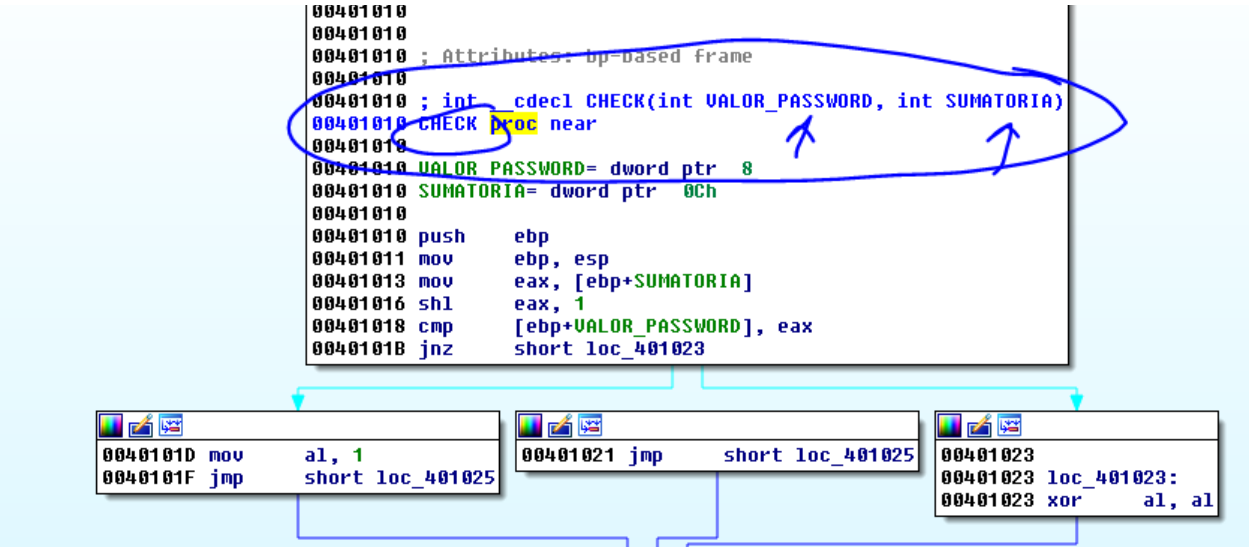
Здесь мы видим два аргумента, очевидно, что тот, который ниже будет ***VALOR_PASSWORD*** так как передается с помощью **PUSH** в стек первым и второй, который кладется следующим, будет суммой, и он будет выше.



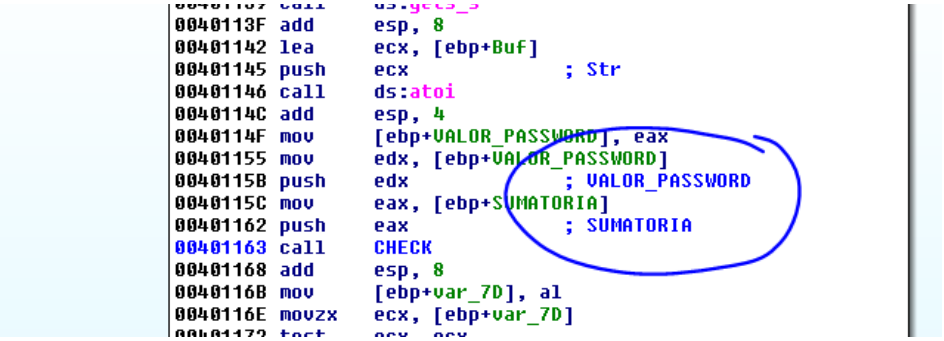
Я переименовываю их согласно этому, и затем, чтобы проверить нормально ли всё, сделайте правый щелчок по адресу ***SUB_0x401010*** и выберите ***SET_TYPE***.



При этом **IDA** попытается объявить функцию со своими аргументами, чтобы показать их в ссылке и мы переименуем также это в функцию ***CHECK***.



И если мы пойдём к ссылке.



Мы видим, что **IDA** распространит имена и говорит мне, что у **EAX** есть ***СЛОЖЕНИЕ*** и **EDX** - это ***VALOR_PASSWORD***.

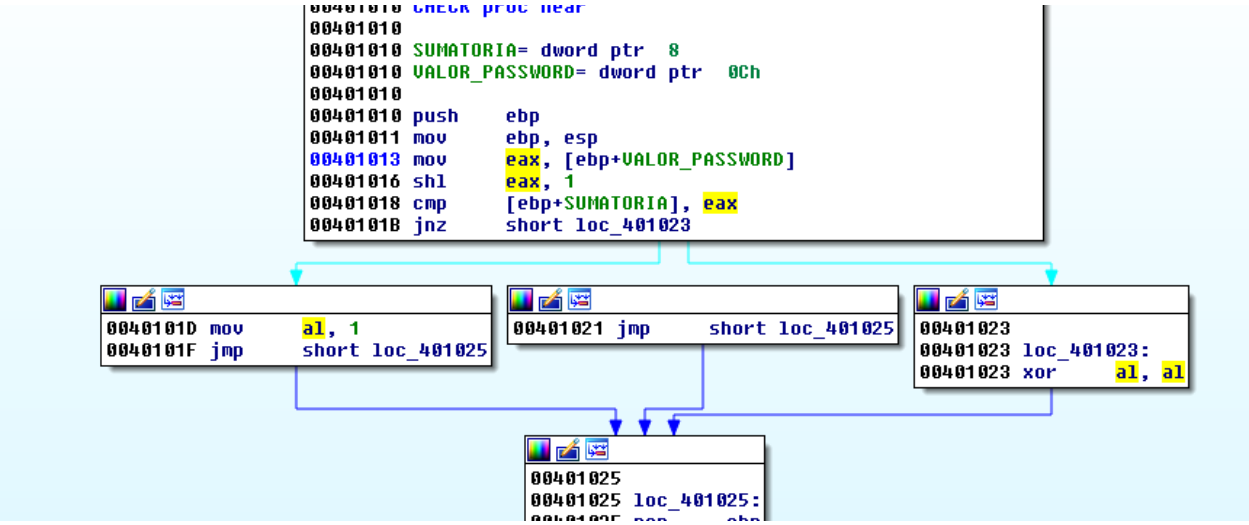
И что сделает функция ***CHECK*** с этими двумя аргументами?

Мы видим, что она сравнивает их, но сначала она берет значение ***PASSWORD*** и делает с ним операцию **SHL EAX, 1**

Examples

`shl eax, 1` — Multiply the value of EAX by 2 (if the most significant bit is 0)

Мы знаем, что **SHL** сдвигает биты влево, заполняя нулями те, которые исчезают на другой стороне, но в частности **SHL *REG*, 1** - это равносильно умножению на **2**.



Программа берет значение пароля, умножает его на **2** и сравнивает его с суммой символов слова ***ПОЛЬЗОВАТЕЛЬ***.

```
Python>ord("p")
112
```

При этом мы получаем числовое значение символа, мы можем сделать формулу, которая суммирует все символы строки ***pepe*,** которую я использую как ***ПОЛЬЗОВАТЕЛЬ***.

```
Python>hex(ord("p")+ord("e")+ord("p")+ord("e"))
0x1aa
```

Мы видим, что сумма равна **0x1AA**, с другой стороны значение, которые мы ввели как пароль будет умножено на **2** перед сравнением с этим **0x1AA**, поэтому правильный пароль должен быть значением, которое при умножении на два даёт нам **0x1AA**.

***X*2 = 0x1AA**

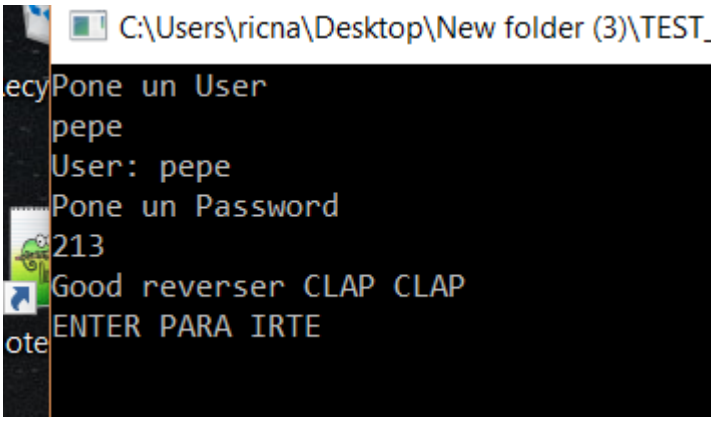
Давайте проясним, что эта программа имеет ограничения, если сложение даёт нечетное число, то невозможно, чтобы существовало значение ***X***, которое при умножении на 2 даёт нам результат нечетное число, поэтому эти имена пользователей не имеют решений , в этой программе имеют решения только имена пользователей, чья сумма при сложении чётная.

Мы очищаем строку и вводим.

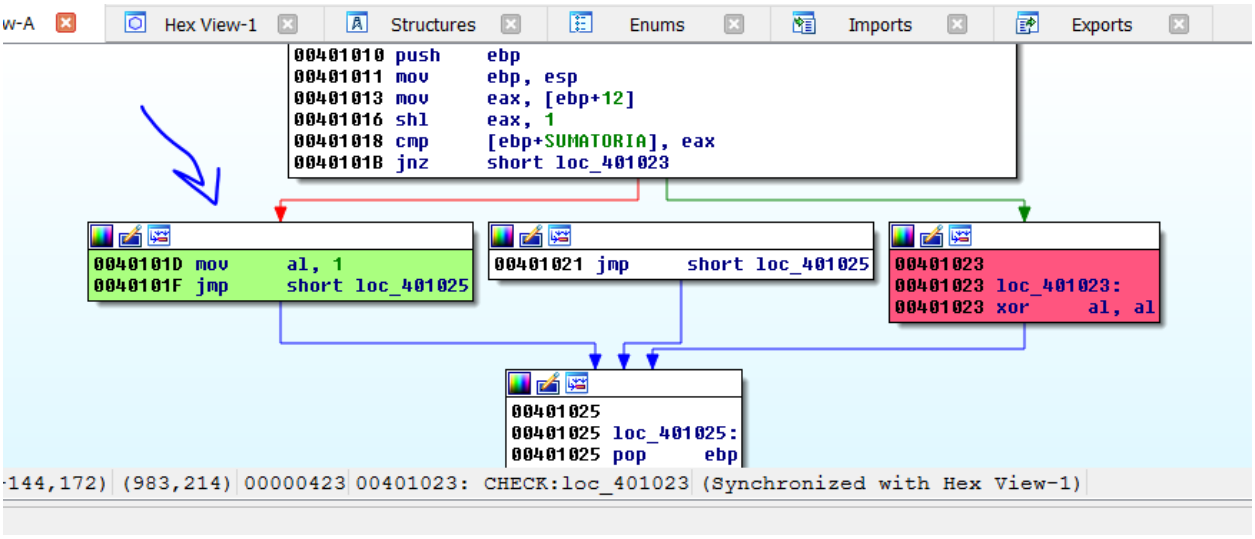
X* = 0x1AA / 2** и ответ поступает в десятичной системе счисления, очевидно, что с помощью ***ATOI он переводится из десятичной системы счисления в ***HEX***.

```
Python>0x1aa/2
213
```

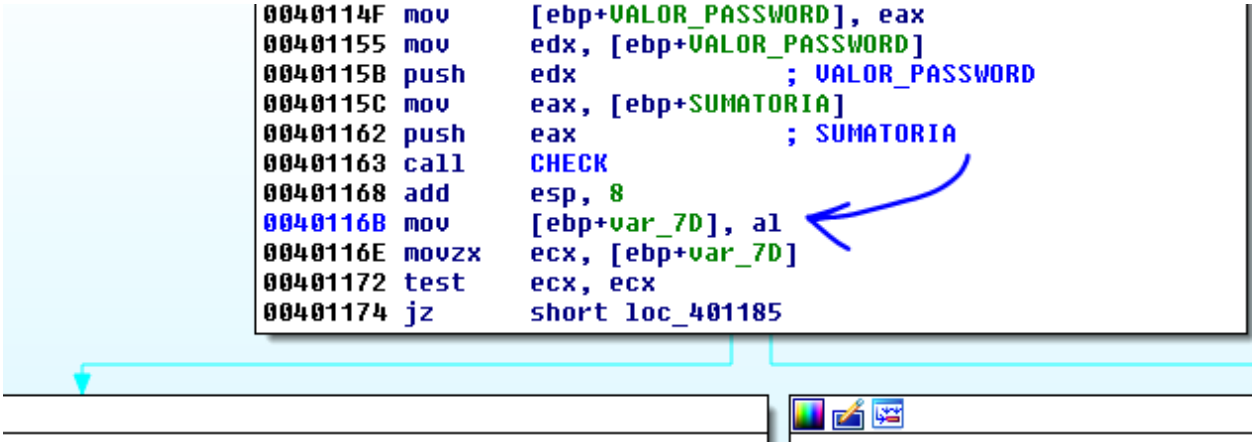
Если я введу ***ИМЯ ПОЛЬЗОВАТЕЛЯ*** как **pepe**, а ***ПАРОЛЬ*** как **213**, что произойдёт?



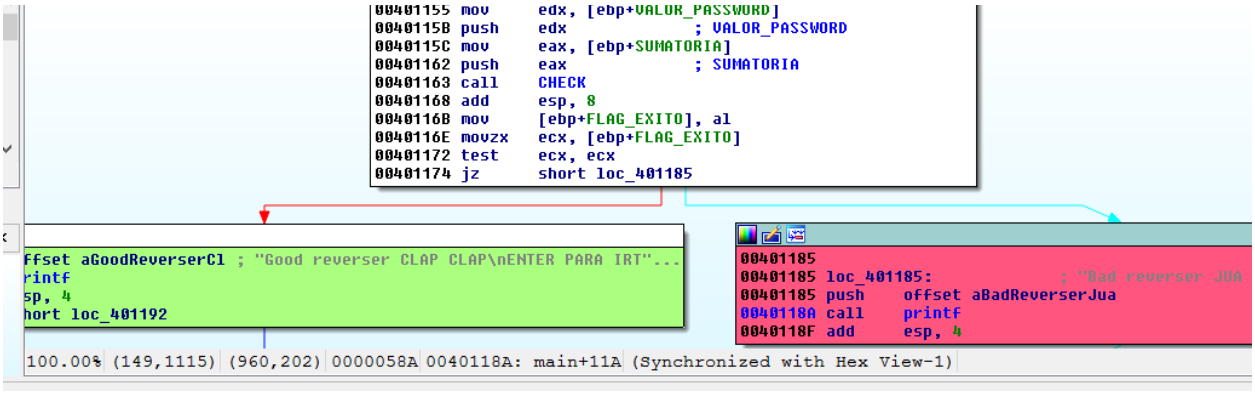
Конечно, я вижу, когда сравнение одинаково внутри функции проверки.



Если они не равны, программа идёт в красный блок и возвращает ***НУЛЬ***, а если они равны программа идёт в зелёный блок и возвращает ***ЕДИНИЦУ***, давайте посмотрим, что произойдёт с этим возвращаемым значением.



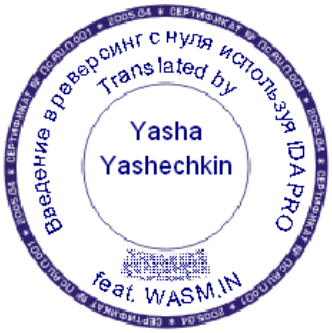
Программа сохраняет его здесь, давайте переименуем это значение в ***FLAG_EXITO***.



Так что, поскольку, если мы видим **0**, то тогда пойдём к ***BAD REVERSER***, а если **AL** равно **1**, то идём к ***GOOD BOY*** как это и случается у нас здесь.

Мне бы очень понравилось, если бы вы отладили этот крэкми и проверили всё, что мы вместе реверсили, установив ***BPs*** и увидев значения в каждом случае до окончательной проверки.

До встречи в **13-*той*** части.



Перевод на английский: **IvinsonCLS**

Перевод на русский с испанского+английского: **Яша_Добрый_Хакер(Ростовский фанат Нарвахи)**.

Перевод специально для форума системного и низкоуровневого программирования - **WASM.IN**

02.09.2017

Версия 1.0

Всем, кому интересна судьба этого и следующих проектов, прошу посетить ресурс ***yasha.su*

Этот, последующие и предыдущие ***PDF*** можно будет в дальнейшем взять на этом ресурсе.